VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmus A* Search implementovaný pomocí technologie CUDA

CUDA Powered A* Search Algorithm

2013 Jan Ivanovský

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Ivanovský**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Algoritmus A* Search implementovaný pomocí technologie CUDA
CUDA Powered A* Search Algorithm

Zásady pro vypracování:

Cílem bakalářské práce je implementovat algoritmus A* Search. Bude implementován za použití běžných programovacích technik, následně pomocí masivního paralelního zpracování technologií NVIDIA CUDA. Součástí práce bude výkonnostní srovnání obou verzí algoritmu.

Práce musí splňovat následující body:
1. Krátký přehled algoritmů pro vyhledávání optimálních tras.
2. Souhrn výhod a nevýhod algoritmu A* Search.
3. Implementace algoritmu A* Search pro CPU.
4. Implementace algoritmu A* Search pro GPU.
5. Srovnání výkonnosti obou implementací.

Seznam doporučené odborné literatury:

[1] Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Pearson Education, 2003.
[2] Judea Pearl. Heuristics: intelligent search strategies for computer problem solving. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
[3] David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series). Morgan Kaufmann, 1 edition, February 2010.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Pavel Dohnálek**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava*.

V Ostravě 6. května 2013 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2013 .....

Rád bych zde poděkoval vedoucímu mé bakalářské práce Ing. Pavlu Dohnálkovi za připomínky a cenné rady.

Abstrakt

Tato práce se zabývá porovnáním rychlosti dobře známého vyhledávacího algoritmu A* na procesoru a grafické kartě. Hledá se ideální rozdělení vláken, kde každé prohledává obecný graf, který byl vytvořený pomocí triangulace. Práce srovná několik způsobů jak náhodně položené body v rovině spojit a ukáže, proč je nejvhodnější Delaunayho triangulace. Výsledné řešení je kontrolováno grafickým znázorněním díky knihovnám OpenCV. Očekává se, že výsledné programy pro počítání algoritmů, at' už na procesoru nebo technologii CUDA, budou velmi podobné. Každé vyhledávání počítá jen čistě A* algoritmus a nebere v úvahu žádné jiné pomocné postupy.

Klíčová slova: A*,CUDA,Delaunayho triangulace,OpenCV

Abstract

This thesis presents a speed comparison of the well know A* Search algorithm implemented on processor and graphics card. The goal is to find the ideal thread division where each thread searches through a common graph created by a triangulation. The work compares several ways of connecting randomly generated points on a plane together and shows why it is best to use Delaunay triangulation. The results are evaluated by graphical representation generated by OpenCV libraries. It is presumed that the final programs for computing algorithms, either on the processor or CUDA, are very similar. Every search only computes the A* algorithm and does not take into account any other auxiliary procedures.

Keywords: A*,CUDA,Delaunay triangulation,OpenCV

Seznam použitých zkratek a symbolů

CUDA	–	Compute Unified Device Architecture
OpenCV	–	Open Source Computer Vision Library
CPU	–	Central Processing Unit
GPU	–	Graphic Processing Unit
ANSI	–	American National Standards Institute
CD	–	Compact Disc

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato bakalářská práce se zabývá implementací a možným použití A* vyhledávání na CUDA a procesoru. Hledá se ideální implementace algoritmu, aby se co nejméně lišil na grafické kartě a na procesoru. Přitom musí být zachována maximální efektivita díky paralelizmu CUDA a výpočetního výkonu pro jedno vlákno procesoru.

Pro řešení se také využívána Delaunayho triangulace. Triangulace vzniká propojením bodů v trojúhelníky. Kdyby se jednoduše propojily body s nejbližšími, tak se nemusí dostat spojitost grafu. A naopak, kdyby se spojily body po řadě, jak se vytváří, tak by se mohly dostávat nepřehledné grafy. Řešením je Delaunayho triangulace, která vhodně spojí nejbližší body a zároveň zachovává spojitost.

V první části práce jsou vysvětleny základní pojmy a jejich možné reálné a teoretické použití. Jsou zde vysvětleny i jiné příklady než byly použity v této práci a teorie okolo jiných vyhledávacích algoritmů.

V druhé části se řeší samotná implementace a pár ukázek kódu. Je zaměřena spíše na použití v programu a je naznačeno hrubé srovnání s podobnými postupy. Také je ukázan postup při vykreslování grafu do obrázku.

Třetí část srovnává výpočetní výkon CUDA a procesoru pro různé řešení.

2 Vysvětlení pojmů a jejich obvyklé použití

2.1 CUDA

CUDA^{TM} je masivně paralelní výpočetní platforma a programovací model vyvinutý společností NVIDIA. Dovoluje výrazný nárůst výpočetního výkonu využitím síly grafické karty. Narozdíl od podobných technologií má technologie CUDA tu výhodu, že grafické kartě předává ke zpracování kód ve známé syntaxi jazyka C/C++ nebo Fortran, tudíž není zapotřebí znalost jiných jazyků. Také dovoluje použití vlastních paralelních algoritmů nebo knihoven pomocí jiných známých programovacích jazyků. V roce 2010, na mezinárodní superpočítačové konferenci v Hamburgu v Německu, představila společnost NVIDIA počítač založený na výpočetním výkonu grafické karty, který se stal druhým nejrychlejším superpočítačem na světě. V roce 2011 dosáhl superpočítač založený na technologii CUDA na první místo a stal se tak pomyslnou špičkou v technologické křivce. Tyto superpočítače také dosahují lepšího poměru spotřeba/výkon než u běžných superpočítačů založených na jiných technologiích. [11]

Z pohledu programátora spočívá CUDA z *počítače - host* a jednoho, nebo více *zařízení-devices*. Počítačem se myslí obecně vzato Centrální procesorová jednotka (CPU), jako například Intel® architektury mikroprocesoru v osobních počítačích. A zařízení je masivně paralelní spojení procesorů vybavených velkým počtem výpočetních jednotek. Srovnání zachycuje obrázek 1 Kód počítače je psán v ANSI C - standartním C a je pouštěn jako normální CPU proces. Kód zařízení je psán také v ANSI C, ale je přidaný o klíčová slova zajišťující funkcionalitu paralelizmu, nazývané *kernely*, a jejich přidružené datové struktury.

Aby mohl programátor spustit kernel na zařízení, musí si přidělit na něm paměť a poslat data z počítače do této přidělené paměti. Po vykonání kernelu potřebuje naopak dostat data zpět a uvolnit přidělenou paměť na zařízení. Nejedná se o obvyklou funkci, která by mohla vracet nějakou hodnotu, ale striktně o $void$. Kernel rozdělí data do jedné *mřížky-grid*. Mřížka se pak dělí na jeden nebo více *bloků-blocks* a každý se ještě rozděluje na stejný počet *vláken-threads*, jak je ukázáno na obrázku 2. Všechna vlákna zařízení spouští stejný kernel, stejný kód. Aby každé vlákno mohlo provádět něco jiného, tak si vlákno zjistí postavení v bloku a postavení svého bloku v mřížce. [2] Ideální velikost mřížky a bloků může určit sám programátor pomocí:

$$dim3\ \ grid(DIM, DIM);$$

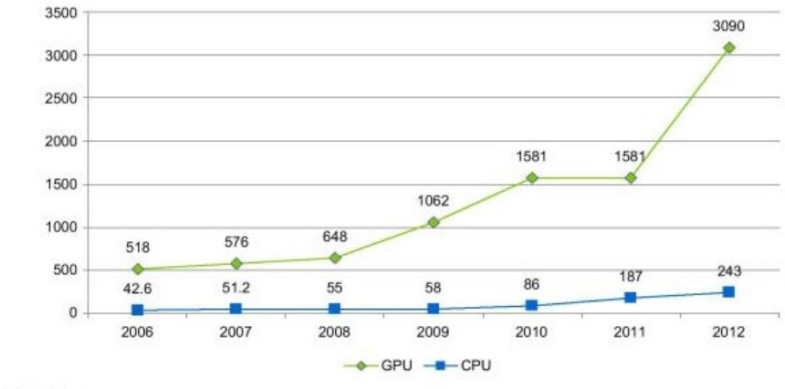
Kde $dim3$ není standartní typ jazyku C. Je to trojrozměrná n-tice, která specifikuje velikost dat programátora. Je také vhodné určit počet vláken v bloku:

$$dim3\ \ threads(DIM2, DIM2);$$

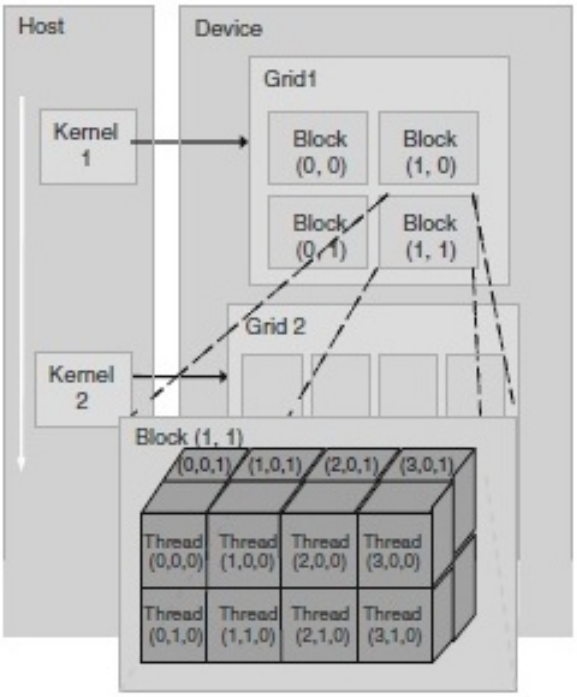
Proměnná DIM určuje kolik bloků bude obsahovat dvourozměrné pole gridu a proměnná $DIM2$ určuje jak velké pole vláken se spustí v každém jednotlivém bloku. Poté se spustí kernel jako:

$$kernel <<< grid, threads >>> ();$$

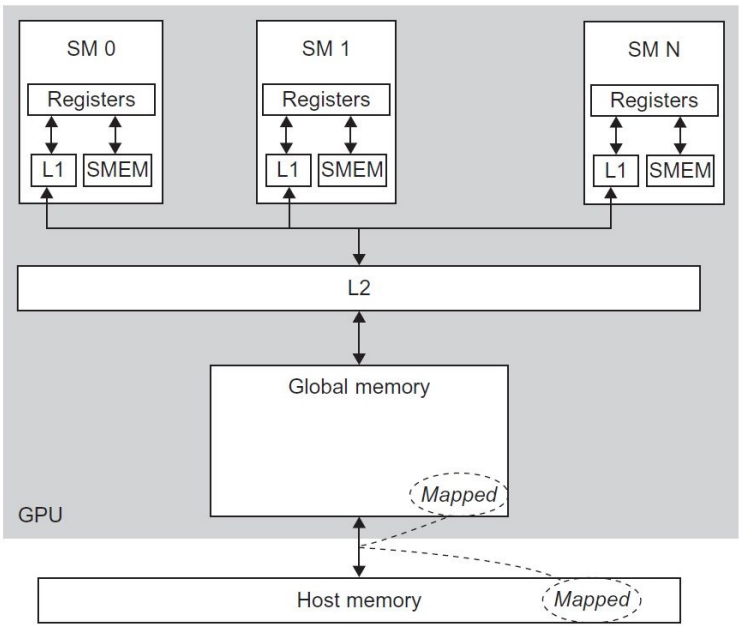
Kernel tak proběhne mnohnokrát paralelně na zařízení podle rozměrů které dostal. S paralelizmem přichází mnoho problémů jako sdílení zdrojů nebo nezaviněné zapisování



Obrázek 1: Srovnání výkonu GPU a CPU v gflops



Obrázek 2: Rozdělení mřížky na bloky a vlákna

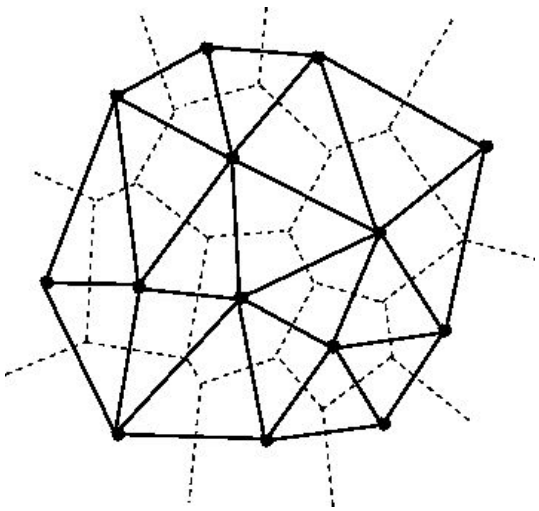


Obrázek 3: Rozdělení paměti

dat na jedno místo více vlákny, a proto se již v kódu ošetří tyto události. Je výhodné psát programy paralelně, a zároveň důležité je psát efektivně. Každé vlákno zjistí zpětně svoje umístění v bloku pomocí $threadIdx$ a umístění bloku v mřížce pomocí $blockIdx$. Dále se data rozdělují do $warp$ o 32 vláknech, toto ale probíhá automaticky. Skupina vláken a skupina warpů se spouští společně, ideálně všechny jednou a společně. Maximální velikost každého rozměru mřížky je 65535. Maximální velikost rozměru x a y bloku je 512 až 1024 podle kompatibilní verze zařízení a rozměru z až 64. Také samotná paměť jednoho vlákna je omezena na 16KB a proto je vhodné aby pracovala na alokované paměti zařízení, než aby vytvářela vlastní rozsáhlá data. Také není možné aby vlákno vytvořilo jiné nebo samotná rekurze. Naproti tomu můžou vlákna spolupracovat synchronizací přes funkci $syncthreads()$. V tomto bodě se všechna vlákna zastavují, dokud ho nedosáhnou všechna. Jakákoliv jiná komunikace mezi vlákny není možná. [1] [12]

Důvod pro alokování dat na zařízení je jednoduchý. Rychlost přístupu k paměti mimo zařízení není dostatečná pro náhodný přístup výpočetních procesorů na technologii CUDA. Přístupový čas do registrů je v řádu TB/s, kdežto ze zařízení do počítače je to v řádu jednotek GB/s. Více je znázorněno na obrázku 3 . Gridy, bloky a vlákna jsou vytvořeny sice kernelem, ale musí se volat z počítače. Tento způsob je jediný, jak výše uvedené struktury udělat. Nemůžou být vytvořeny uvnitř samotné funkce kernelu. [7]

CUDA technologii využívají i společnosti jako NASA, Adobe nebo Wolfram Research. Příkladem mohou být věrohodné simulacé písku, interaktivní simulace deformovatelných objektů v reálném čase, doostření fotografie, simulace černé díry, zaznamenávání



Obrázek 4: Aplikace Delaunayho triangulace na Voroniovy diagramy

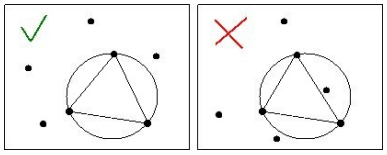
pohybu ze stovek kamer v jednu chvíli, zobrazení molekul, analýzy letového provozu nebo hledání skrytých vad v tepnách člověka a mnohé další. [5] [6]

2.2 Delaunayho triangulace

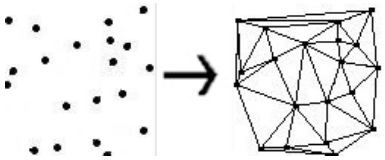
Při triangulaci množiny bodů je žádoucí, aby vytvořené trojúhelníky byly co nejvíce rovnostranné. Po sestrojení těchto trojúhelníků by měl každý co nejlépe lokálně reprezentovat hodnotu povrchu. Další požadovaná charakteristika triangulačního procesu je, aby byla produkována jednoznačná triangulace nezávisle na počátečním bodě nebo orientaci množiny dat. Tyto výsledky budou předpověditelné a jednoduše opakovatelné. Sibson v roce 1978 ukázal, že Dalaunayho triangulace tyto podmínky obecně splňuje, i přes to, že jsou určité konfigurace množiny dat, které mají na výstupu lokálně nejednoznačné řešení.

Delaunayho triangulace je blízce příbuzná k Dirichletově mozaikování (Dirichlet tessellation) množiny bodů, které rozdělí body unikátní množinou polygonů, které se nazývají Thiessenovy polygony, nebo také Voroniovy diagramy. Thiesenovy polygony ohradí všechny body oblastí, ve kterých jsou všechna místa bližší k danému bodu než k jinému bodu z dané množiny bodů. Každá hrana každého polygonu je kolmá osa oddělující ohrazený bod od sousedních bodů sousedním polygonem. Když jsou všechny sousední body propojeny hranami, Delaunayho triangulace končí, viz. obrázek 4.

Nebude se v této práci vycházet z Thiesenových polygonů, ale jednoduše z bodů v rovině. Algoritmus náhodně vybere tři body a proloží jimi kružnici. Pokud uvnitř této kružnice žádný jiný bod neleží, tak se vytvoří trojúhelník. V případě že v kružnici je bo obsažen, tak algoritmus vybere jiné tři body. Tohoto postupu bylo také použito na obrázku 5 a na obrázku 6 . Nebo lze také udělat obecnější algoritmus, který hledá jakoukoliv prázdnou kružnici mezi dvěma body.



Obrázek 5: Kružnice, která určuje, zda může trojúhelník mezi body vzniknout



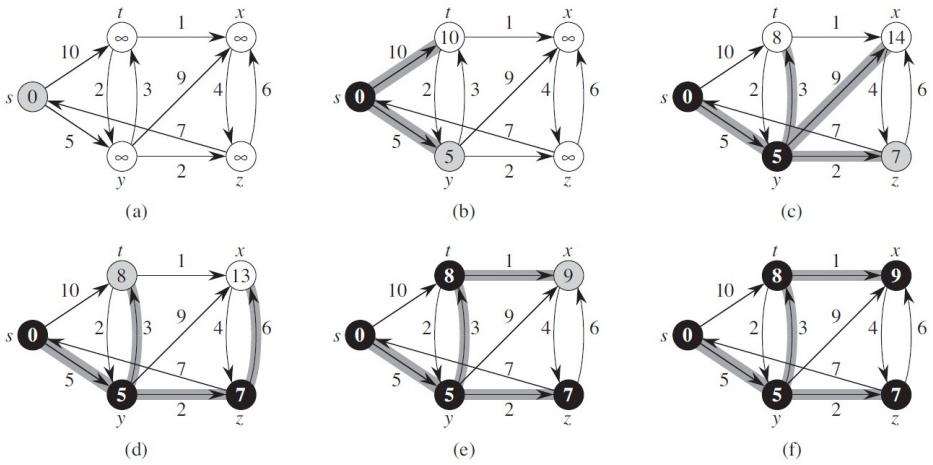
Obrázek 6: Použití triangulace na body v rovině

Další vlastností Delaunayovy triangulace je, že maximalizuje minimální úhel. To je nejmenší úhel ze všech úhlů v trojúhelnících z triangulace. Tento úhel je větší při použití Delaunayovy triangulace, než při použití jakékoliv jiné metody, a to pomáhá pro tvorbu přehlednějšího grafu. Delaunayova triangulace je také jednoznačná, pokud neobsahuje singulární případy bodů, kdy více bodů leží na společné kružnici opsané. [3]

2.3 Vyhledávací algoritmy

Vyhledávací algoritmy se můžou dělit různými způsoby. Například se výrazně liší způsobem hledání, ale i možným výsledkem. Hledání samotné bývá pro některé počítačové programy nejsložitější operací, jakou provádí. Mohou například hledat jeden prvek nebo hledat cestu k němu. Zde nastává problém, protože chce-li program najít nejkratší cestu z bodu A do B, první nalezená cesta nemusí být nutně nejkratší a nemusí mít jednoznačný výsledek. Nebo naopak vyhledání nejdelší možné cesty, nebo problém obchodního cestujícího, kdy se snažím projít celý graf bez opakování uzlů. Například vyhledávání hrubou silou najde v konečném čase nejkratší cestu, ale zkoumá všechny možnosti, kterých může být i pro malý prostor neúměrně mnoho. [8]

Algoritmus prohledávání do hloubky na grafech má pojmenování podle vybírání prvků, které byly naposledy prohledávány a tak se dostane velmi rychle daleko od počátečního bodu. Jakmile nalezne konečný bod, tak skončí i když není cesta ideální. Podobný přístup má Dijkstrův algoritmus, který v každém cyklu přidává do prohledávané podmnožiny uzlů právě jeden uzel s nejmenším ohodnocením. Na obrázku 7 lze vidět použití algoritmu, kde šedé hrany označují předchůdce, černé uzly jsou zahrnuty do množiny projdených uzlů. Poslední část (f) vyjadřuje všechny minimální cesty z uzlu s. Protože prohledává všechny prvky, které můžou mít kratší cestu než již nalezenou, tak konečná nalezená cesta je nejkratší. Bouzel se jedná o prohledávání naslepo a i když algoritmus najde nejkratší cestu, tak může ještě prohledávat dál, nebo i když je blízko konečného uzlu, tak může zbytečně prohledávat jiné cesty. Na rozdíl od bellman-ford algoritmu, nedokáže efektivně využívat záporně ohodnocené grafy. Bellman-ford algoritmus dokáže



Obrázek 7: Příklad použití Dijkstrova algoritmu na ohodnoceném grafu.

detekovat v grafu záporně ohodnocené cykly, to je cyklus v grafu, jehož součet hran má hodnotu menší než 0. Najde-li takový, tak jakákoli cesta může být kratší než vzdálenost 1 a hledání optimální cesty tak pozbývá smyslu. [15] [10]

Naopak hladový algoritmus vybírá prvky, které vypadají nejlépe v daný moment. Je založen na vyřešení lokálního problému, tak ale nemusí vyřešit problém globální. Nalezená cesta nemusí být optimální, i když pro určité problémy obvykle je. Na rozdíl od Dijkstrova algoritmu se snaží dostat do konečného bodu co nejrychleji a když nalezne jakoukoli i neoptimální cestu, tak algorimus končí. Nalezne tak cestu z počátečního do konečného bodu rychleji, ale za cenu jistoty, že je cesta optimální. [9]

2.4 A* algoritmus

A* je vyhledávací algoritmus, který byl vytvořen v roce 1968. Kombinuje Dijkstrův algoritmus a heuristický prvek - hladový algoritmus. Heuristický obvykle vyjadřuje přibližný způsob, jak řešit problémy, aniž by bylo zaručeno, že se najde nejlepší odpověď. A přesto, že je A* algoritmus postavený přímo na heuristickém prvku, tak zaručuje nejkratší nalezenou cestu.

Při vyhledávání mezi dvěma body využívá Dijkstrův algoritmus, který upřednostňuje uzly blíže počátku a A* se snaží o redukci prohledávaných uzlů, a proto bere informace z hladového heuristického hledání, které upřednostňují uzly blíže cíli. Ve standartní terminologii používané A*, $g(n)$ vyjadřuje přesnou cenu cesty z počátku do daného uzlu n, a $h(n)$ vyjadřuje heuristicky spočítanou cenu z daného uzlu n do cíle. To je minimální délka teoreticky nalezené cesty z daného uzlu, kde optimální cesta je delší, nebo rovna $h(n)$. A* poté vždy pro počítání vybere uzel, který má nejnižší cenu $f(n) = g(n) + h(n)$. Heuristický spočítaný prvek, nikdy nesmí podhodnotit teoreticky spočítanou cestu do cíle. A* algoritmus je také kompletní,a to ve smyslu, že nalezne cestu, pokud nějaká

existuje, respektive když jsou si počáteční a konečný uzel navzájem vždy dostupné. Například, pokud uzly jsou body v dvourozměrném prostoru se souřadnicemi x_i a y_i, a jedná se o Eukleidovský prostor, pak platí:

$$h(i) = \sqrt{[(x_i - x_t,)^2 + (y_i - y_t)^2]}$$

Kde $h(i)$ je následně nejkratší možná vzdálenost z bodu i do bodu t . [14]

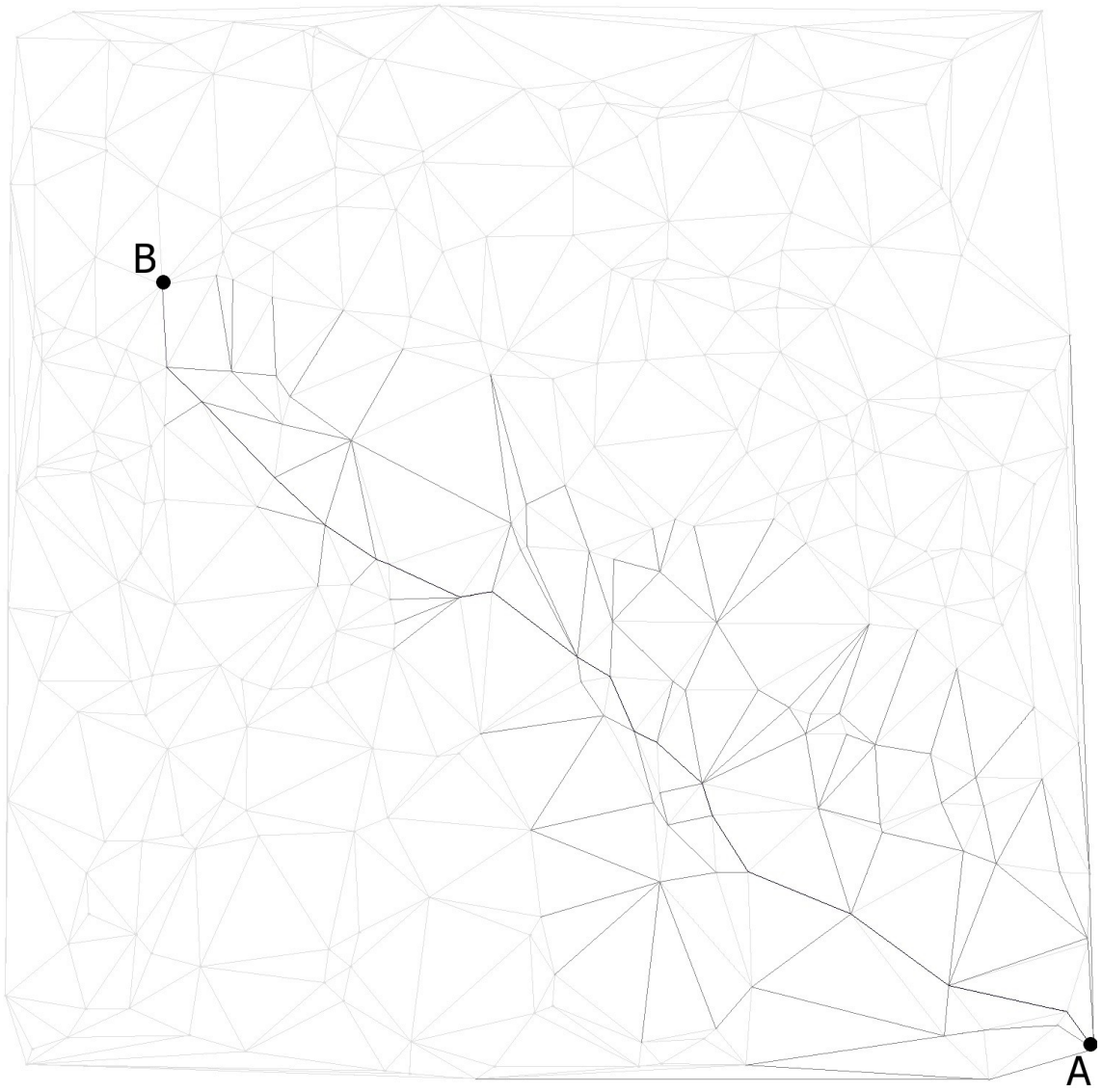
Pokud budou cesty ohodnoceny přesně jako heuristicky spočítaná cena, pak není potřeba kontrolovat nahrazování již nalezené cesty. Je to dáno tím, že uzel se nemůže tak zhoršit, aby pak do něj byla nalezena levnější cesta. Nicméně, heuristický prvek neuvažuje překážky v grafu. Jakmile prohledávání narazí na nějakou překážku, tak se může cena cesty už jen zvětšovat při stejném heuristickém prvku. Doba hledání stoupá, když se snaží algoritmus překážku obejít - to nemusí být možné. Do algoritmu lze přidat prvky hrubého prohledávání dopředu, které by hledaly překážky a poté by bylo možné určit zdali je vhodné začít prohledávat v uzavřeném prostoru. Zvláště, když se z tohoto prostoru lze dostat jen jednou cestou a tento prostor neobsahuje konečný bod. [4] [13]

V příloze je ukázka pod názvem $AStar.gif$

Pro použití v reálném čase má A* dvě nevýhody. První nevýhodou je potřebný výpočetní čas pro nalezení cesty z počátečního bodu do konečného. Druhou je spotřebování velkého množství paměti během běhu algoritmu. To znemožňuje použití velkého množství vyhledávání v jednu chvíli, protože každé vyhledávání potřebuje udržovat $h(n)$,$g(n)$ a $f(n)$ pro každý procházený uzel n. V řešení programu bylo použito pole pro zapamatování informací o $h(n)$ a $g(n)$, jestli je uzel procházený a sousední uzel ze kterého přišla nejkratší cesta $g(n)$. [5]

Jiný způsob řešení je prioritní fronta otevřených, tj. ještě nenavštívených uzlů. Jakmile jsou všechny sousední uzly takto otevřeného uzlu prohledány, změní se stav uzlu na "uzavřený"nebo "neaktivní"a je odstraněn z fronty otevřených uzlů. Každý jeden cyklus programu jeden uzel odstraní z prioritní fronty a naopak jich několik přibude. Ve frontě jsou uzly seřazeny podle hodnoty $f()$ a je vybírán pro počítání vždy ten s nejnižší hodnotou. Algoritmus začíná tak, že se přidá počáteční uzel do prioritní fronty a prohledají se jeho sousední uzly. Poté je z fronty odstraněn a vybere se další uzel s nejnižší hodnotou $f()$.

Na obrázku 8 lze vidět všechny zkoušené cesty s bodu A s jedinou vyznačenou optimální cestou do bodu B. Všechny takové cesty vyjadřují hodnotu $g(n)$ z počátečního uzlu.



Obrázek 8: Prostor procházených cest z bodu A do bodu B pomocí A* algoritmu

3 Implementace zadání a ukázky kódu

3.1 Implementace Delaunayho triangulace

Bylo zvoleno řešení Delaunayho triangulace pomocí kružnice opsané třem vybraným bodům. Nejprve byly zvoleny tři body body a poté se prochází všechny ostatní body, zda-li nejsou v kružnici opsané a to díky testu nerovnosti:

$$(x - x_0)^2 + (y - y_0)^2 - r^2 \leq 0$$

, kde x a y jsou souřadnice zkoušeného bodu, x_0 a y_0 je střed kružnice a r je poloměr kružnice. Tato rovnice je ovšem pro otestování všech bodů výpočetně složitá. Na výpočet je jednoduší vypočítat hodnotu determinantu:

$$\begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix}$$

, kde A, B a C jsou tři body vybrané pro kružnici opsanou a bod D je zkoušený bod. Zjišťuje se, zda tento bod neleží uvnitř kružnice. Při testování je důležitá orientace bodů A, B a C zda jsou psány po směru hodinových ručiček v kružnici opsané, nebo naopak. Podle toho vyjde determinant kladný, nebo záporný vzhledem k vnějším/vnitřním bodům. Proto se jednoduše otestuje, zdali je střed kružnice vepsané kladný nebo záporný pro výsledek determinantu.

$$\begin{gathered} (((B_y - D_y) * (C_x * C_x - D_x * D_x + C_y * C_y - D_y * D_y)) + \\ +((A_y - D_y) * (C_x - D_x) * (B_x * B_x - D_x * D_x + B_y * B_y - D_y * D_y)) + \\ +((B_x - D_x) * (C_y - D_y) * (A_x * A_x - D_x * D_x + A_y * A_y - D_y * D_y)) - \\ -((A_x - D_x) * (C_y - D_y) * (B_x * B_x - D_x * D_x + B_y * B_y - D_y * D_y)) - \\ -((B_x - x1) * (A_y - D_y) * (C_x * C_x - D_x * D_x + C_y * C_y - D_y * D_y)) - \\ -((C_x - D_x) * (B_y - D_y) * (A_x * A_x - D_x * D_x + A_y * A_y - D_y * D_y)) \\ > 0) \end{gathered}$$

Také může nastat situce že bude determinant pro náhodný bod D nulový, pak tento bod leží na kružnici opsané. Tak je porušena jednoznačnost Delaunayho triangulace a nabízí se více řešení, jak body A, B, C a D spojit.

3.2 A* na procesoru

Pro spuštění programu bylo pochopitelně zvoleno řešení s vlákny. Bez vláken trvá běh programu průměrně dvakrát delší dobu při málo uzlech. Při více jak 500ti uzlech je již rozdíl nepatrný kvůli náročnosti programu a vytížení jednoho vlákna. Jedno vlákno dostane jeden uzel a z něj počítá do každého uzlu optimální cestu. Do pole se zapíšou vzdálenosti u příslušných prvků a jen u počátečního a končeného uzlu se zapíše první sousední uzel kam se vydat, pokud se hledá nejkratší cesta z počátečního do konečného

uzlu a naopak. Toto se děje pro každý uzel s každým uzlem a tak vnikne pole pro graf, kdy každý uzel zná nejkratší cestu do jakéhokoli uzlu. Celý proces je paměťově náročný a každé hledání potřebuje své vlastní pole procházených uzlů, jejich vzdáleností od počátku a konečného uzlu. Program neprobíhá moc paralelně, a proto spotřeba paměti pro pár vláken není velká. Vlákna také mezi sebou efektivně nespolupracují, aby byl zachován koncept algoritmu A*. V případě, že by vlákna mezi sebou spolupracovala, tak jen první vyhledávání by splňovalo požadavky A* algoritmu a ostatní by postupně získávala nerovnovážný heuristický prvek.

A* algoritmus k sobě potřebuje pole hodnot, kam si může ukládat hodnoty pro počítání funkce $f(n) = g(n) + h(n)$ pro každý uzel/prvek. Také si potřebuje pamatovat, zda byl již uzel procházený a předchozí uzel, ze kterého přišla nejmenší možná hodnota $g(n)$ vzhledem k počátečnímu uzlu. V případě, že prohledávaná cesta dorazí do uzlu, který má sice již zapsanou hodnotu $g(n)$, ale může se nahradit výhodnější, tak hodnota $g(n)$ sníží a uzel se označí za dosud neprocházený. Zařadí se tak do fronty krajních bodů, které se vybírají pro další hledání. Ovšem tato možnost nahrazení lepší možností neprobíhá často, protože body jsou pospojované triangulací, která neobsahuje překážky, a ohodnocení sousedních uzlů je vždy rovno heuristickému prvku. Přesto je tato možnost nahrazení naimplementována. Také to znamená, že první nalezená cesta z počátečního do konečného uzlu je také optimální cestou.

Druhý způsob jak vybírat prvky pro procházení je vytvoření fronty/seznamu, ve které se porovnávají výsledky $f(n)$. Tento způsob řešení může být zdlouhavý vhledem k poměru počtu vkládání k počtu vybírání. Každý vkládaný prvek se musí zařadit na správné místo a při vybírání se odstraňuje první prvek. Bylo zvoleno řešení s polem, ve kterém se označují aktivní prvky a poté se jedním průchodem vybere s nejmenší hodnotou $f(n)$.

Program procházení grafu probíhá následujícím způsobem:

1. Vytvoření pole, které vypočítá heuristický prvek pro každý uzel a označí každý uzel jako neprojdený.
2. Prochází se uzel s nejmenší hodnotou $f()$. Pokud se zjistí že má za souseda konečný bod, tak se pro něj vypočítá $f()$ s nulovým heuristickým prvkem.
3. Prochází se všechny uzly okolo uzlu s nejmenší hodnotou $f()$, změní se pro ně $g()$ a aktivují se.
4. Vybere se nový prvek s nejmenší hodnotou $f()$, pokud ta je menší než $g()$ u konečného uzlu, nebo pokud konečný uzel tuto hodnotu ještě nemá.
5. Pokud se takový prvek našel, tak se pokračuje od bodu 2.
6. Zapíše se informace o optimální vzdálenosti $g()$ do konečného a počátečního uzlu.

Navíc se zapisují informace o předcházejícím uzlu, ze kterého přišla nejkratší vzdálenost $g()$. Celý cyklus neprobíhá pro identické uzly a také neprobíhá dvakrát pro určité dva

uzly, protože by se stejně řešení opakovala. Dále se nemusí aplikovat vyhledávání na sousední cesty, zbytečně by tak prodlužovaly procházení grafu.

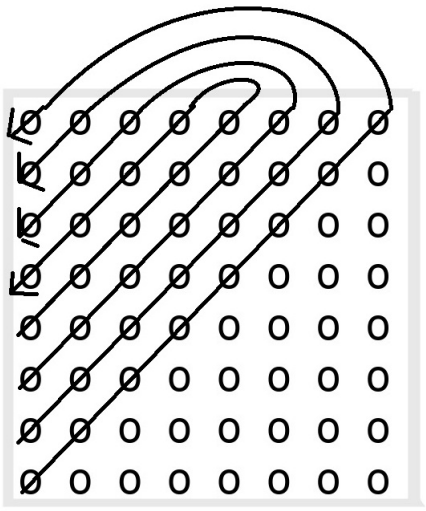
Další funkce využívá dat, které se zapsaly a může tak určit vždy optimální cestu z uzlu do jakéhokoli jiného. Protože když se prochází graf, tak si vždy uzel pamatuje svoji optimální cestu kam může pokračovat a podmínky se nemění, takže se ani optimální cesty nemění. Teoreticky by samozřejmě šlo si zapamatovat přes optimální procházené uzly a do nich si postupně zapsat všechny optimální vzdálenosti, ale pak by se aplikoval A* algoritmus jen na prvních pár vyhledávání a celý graf by byl již skoro zaplněný výsledkem optimálních hodnot. Takové řešení ale nebylo použito, přestože by velmi zvýšilo rychlost prohledání celého grafu a nalezení všech optimálních cest ze všech uzlů do každého jiného.

3.3 A* na CUDA

Stejně jako na procesoru, i technologie CUDA hledá každou nejkratší cestu z každého bodu, aniž by si pomáhala mezivýsledky. Na rozdíl od procesoru byla provedena tři různá řešení. V každém se spustí všechna vlákna najednou a pokud každé počítá jedno vyhledávání, tak už při 400 uzlech se potřebuje teoreticky alokovat přes 1GB dat. Z testování ale vyplynulo, že je lepší nechat vlákno alokovat pole pro první řešení až při jeho spuštění, než posílat jedno velké do zařízení před spuštěním kernelu. Je to dáno nejprve tím, že vlákna přistupující do jednoho velkého pole mají problém s daty, které jsou příliš ”vzdálené”od sebe. Naopak když si každé vlákno alokuje vlastní pole, tak má data ”blízko”od sebe. Pak je to dáno tím, že se nemůže poslat do zařízení pole přesahující velikost kapacity paměti. Když se spustí tisíce vláken najednou, tak se některá spustí virtuálně a hned si nevytvoří pro sebe pole dat a čekají na spuštění. Je možné počítat pro technologii CUDA i řešení která by teoreticky neměla projít, protože přesahují rámce paměti. V prvním řešení se tak vytvoří stejná velikost pole jako pro procesor pro každé vlákno a stejně jako pro procesor se zapíše do něj nejprve heuristický prvek a všechny uzly se označí za neprojdené. Tato operace nezabírá výrazně moc času a většinu času stráví vlákna v samotném procházení grafu a vyhledávání optimální cesty.

Druhé řešení na technologii CUDA počítá jen cesty z prvního uzlu a vlákna nespotřebují tolik paměti jako při prvním řešení. Ovšem pro způsob s alokací pole pro každé vlákno dojde rychle kapacita paměti, to je 16kB pro každé vlákno. Pro způsob alokace jednoho společného pole trvá příliš dlouho jeho alokace vzhledem k době počítání. Naopak na procesoru toto vyhledávání spíše přidává na rychlosti, a proto je průměrně stejně rychlé. Zařízení v tomto případě nemůže využít naplno paralelizmu počítání a většinu času stráví nad alokací pole a vyhledávání hodnot v poli.

Nabízí se tedy kombinace prvního a druhého řešení. Tím je vytvoření 4000 vláken, které postupně projdou všechny kombinace uzlů. Aby nějaké vlákno nepočítalo více uzlů, než jakékoli jiné vlákno, tak všechna vlákna počítají průměrně stejně výpočtů. Prochází pole, jak je naznačeno na obrázku 9 a na obrázku 10 využívá tak efektivně většinu vláken. Pro způsob alokace pole pro každé vlákno je řešení průměrně dvakrát rychlejší než obdobné řešení na procesoru. Nicméně stále je omezení do tisíce prvků.Poté je potřeba alokovat jedno velké pole, které ale nemusí mít velikost pro každou možnou kombinaci,



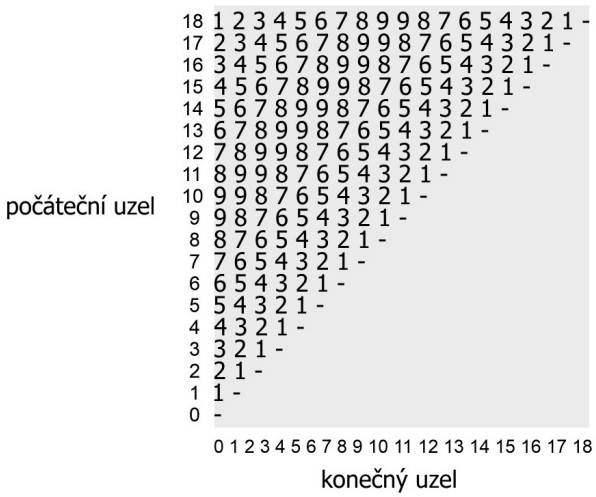
Obrázek 9: Způsob procházení pole

ale jen pro 4000 vláken. Tak se může počítání na technologii CUDA dostat do mnohem více prvků přičemž je stále rychlejší než procesor.

4000 vláken je maximem pro procházení celého pole a alokování společného pole pro vlákna. Pro větší počet vláken se vlákna začnou spouštět i virtuálně, což by obvykle nevadilo, ale některá vlákna se spustí až příliš pozdě a celkově brzdí program.

```
int y1 = (blockDim.y * blockIdx.y + threadIdx.y);
if ( y1 > 64 ) return;
int x1 = (blockDim.x * blockIdx.x + threadIdx.x);
if ( x1 > 64 ) return;
int pomy=y1*64+x1; //pamatuje si vsechny kombinace 0-4096
if ((pomy)>4000) return; //nad 4k vlaken se neresi
int pomx=(pomy/(rowCount/2)); //urcuje kolik pocatecnich uzlu se bude resit najednou
int pr=((pv)/(rowCount/2)); // urci kolikrat jedno vlakno vyhleda cest, tj pocet rozdeleni pole
if (pomx>(pr-1)) return; //aby ukazatel nesel mimo alokovane pole
for (int pomz=0;pomz<((rowCount/(pr))+1);pomz++) //od 0 po
  {
    int x=(pomx)+pomz*(pr); //celkove meneni, vraci hodnotu az do rowCount po cyklech
        0-15, 16-31, 32-47, 48-64 az 484-500 pro rowCount=500
    int y=(pomy%(rowCount/2))+x; //neustale se menici cislo modulo polovinou pole plus az
        rowcount 0-249(1-250....); az 500-749
    if (y>rowCount) //v pripade ze je y nad limit , tak se otoci prubeh
    { //algoritmus se v poli vraci na zacatek pro hodnoty, ktere vynechal
      x=(x+((rowCount/2)-x)*2); //x je ted x+rowcount-y, ktere je vetsi nez rowcount, treba
          -0 az -250 pro 484-500
      y=-y+rowCount+rowCount+1; // -600+500+500+1 =401 treba pro rowCount=500
    }
```

Výpis 1: Ukázka tvorby a použití 4000 vláken



Obrázek 10: Způsob procházení pole, kde čísla v poli vyjadřují číslo vlákna, které kombinaci uzlů počítá

3.4 Zobrazení průběhu algoritmu díky OpenCV

OpenCV je volně šířitelná svobodná knihovna zaměřená na počítačové vidění a zpracování obrazu. Tato knihovna je použita v programu pro zobrazení procházeného prostoru a optimální cesty. Pro zobrazení celého postupu A* algoritmu je zpracování a vytváření obrázků výpočetně a paměťově náročné. Jedno hledání vytvoří desítky až stovky obrázků a může se zdát, že se program zastavil.

Zobrazení jednotlivých bodů je poměrně triviální záležitost, protože stačí vyznačit pixel na daných souřadnicích, případně ještě pár pixelů okolo. Naopak vytvoření úseček pro jednotlivé cesty není tak jednoduché a existuje několik způsobů. Nabízí se nejjednodušší řešení a to zobrazit jen pixely, které jsou přímo na spojnici mezi sousedními uzly. Takových bodů je ale velmi málo a k algoritmu se musí přidat určitá tolerance ať už zaokrouhlení na celá čísla nebo tolerance v rámci procent nebo tolerance činící ± 1 případně kombinace předchozích. Takové způsoby jsou vhodné buď pro stejné vzdálenosti mezi uzly, nebo zobrazení cest se stejnou směrnicí přímky.

Jiný způsob je vykreslování jednotlivých bodů mezi uzly, kdy se začne vykreslovat v souřadnicích jednoho uzlu a souřadnice pro vykreslení se mění po jednom pixelu. Tak je jisté že výsledná úsečka nebude mít více bodů v průměru, nebo nebude ani přerušovaná. Zbývá jen dořešit způsob vyhledání přímé cesty mezi uzly. Nabízí se použít A* algoritmus, nebo podobný, takové řešení ale zabírá moc paměti a je teoreticky nutné alokovat další paměť navíc. Jednodušší hladový algoritmus sice najde nejkratší cestu, ale nejprve postupuje diagonálně a následně mění buď x, nebo y. Výsledné spojnice proto vypadají "zlomeně"než jako přímé spojnice bodů.

Nakonec byl vybrán algoritmus, který bere v úvahu směrnice přímek mezi uzly. Respektive vybírá body, které mají co nejpodobnější poměr stran obdelníků mezi uzly

a mezi uzlem a zkoumaným bodem vzhledem k osám souřadnic x a y. Jak je ukázáno na výpisu programu 2, kde se hledá další bod pro vykreslení mezi uzly, kdy souřadnice počátečního uzlu jsou menší než souřadnice cílového uzlu. Po proběhnutí se vykreslí bod na souřadnicích $bodx$ a $body$ a cyklus se opakuje dokud se body nerovnají souřadnicím cílového uzlu.

```
float d1=(((float)bodx+1−poleu02[i][0])/((float)body+1−poleu02[i][1]));
float d2=(((float)poleu02[j][0]−poleu02[i ][0]) /(( float)poleu02[j][1]−poleu02[i ][1]) ) ;
float d3=(((float)bodx−poleu02[i][0])/(( float)body+1−poleu02[i][1]));
float d4=(((float)bodx+1−poleu02[i][0])/((float)body−poleu02[i][1]));
if (((abs(d2−d1))<(abs(d2−d3)))&&((abs(d2−d1))<(abs(d2−d4))))
{  //pokud je nejlepsi derivace d1 pro bod x+1 a y+1
   bodx++;
   body++;
}  //tak souradnice bodu pro vykresleni se zmeni na x+1 a y+1
else
if (((abs(d2−d3))<(abs(d2−d1)))&&((abs(d2−d3))<(abs(d2−d4))))
{  //pokud je nejlepsi derivace d3 pro bod x a y+1
   body++;
}  //tak souradnice bodu pro vykresleni se zmeni na x a y+1
else
{  //pokud je nejlepsi derivace d4 pro bod x+1 a y, nebo jsou derivace stejne
   bodx++;
}  //tak souradnice bodu pro vykresleni se zmeni na x+1 a y
```

Výpis 2: Ukázka tvorby cest mezi uzly

4 Srovnání výpočetního výkonu procesoru a CUDA

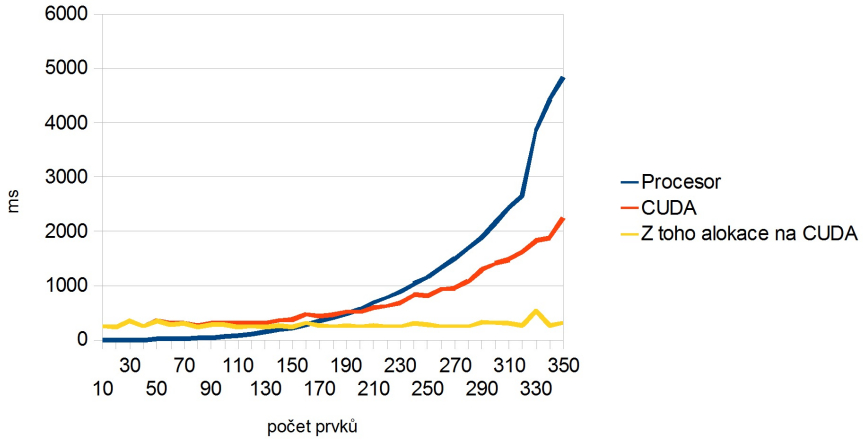
Na obrázku 11 lze vidět, že přibližně do 200 prvků je počítání na technologii CUDA pomalejší jen díky dlouhé alokaci pole na zařízení, přestože je teoreticky rychlejší. Proto není vhodné používat takový postup v programech, kde je důležitá spíše rychlá odezva programu uživateli, než složitost celé operace pro jednotlivé vyhledávání.

Stejný způsob byl použit pro řešení na obrázku 12, kde alokace pole na zařízení je stále okolo 300ms, ale pro více prvků je to již zanedbatelné zdržení. Teoreticky by neměla data pro technologii CUDA projít nad 500 prvků, kvůli kapacitě paměti na grafické kartě 1,5GB, ale díky virtualizaci vláken neproběhnou všechna alokace pole pro jednotlivá vlákna v jednu chvíli a může počítání přesáhnout i tuto hodnotu. Srovnání časů je zapsáno v tabulce 1 v třetím sloupci pod označením *CUDA 1*. Řešení lze použít spíše pro méně prvků, kdy není problém s nedostatkem paměti.

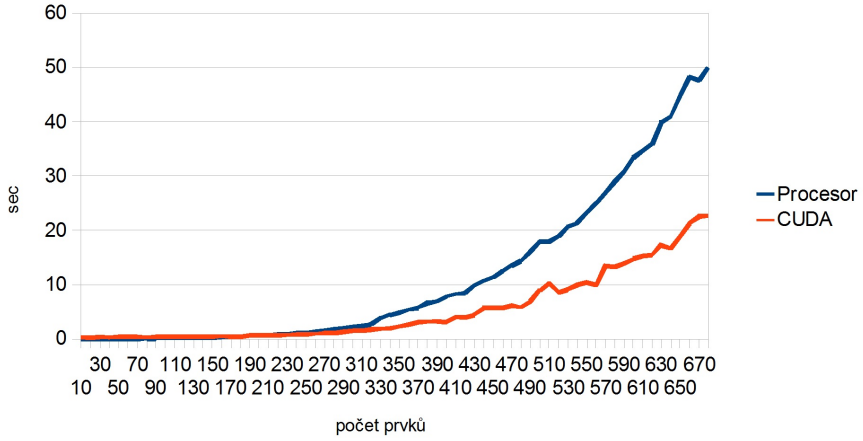
Poslední graf na obrázku 13 vyjadřuje srovnání s použitím 4000 vláken na technologii CUDA. Efektivnější řešení přináší alokace polí přímo ve vláknech, ale takové pole nemůžou mít velikost pro více jak 1000 uzlů. Z tabulky 1 je vidět, že je řešení pod označením $CUDA2$ o něco efektivnější než předcházející. Nakonec bylo použito jedno větší společné pole, které ale nemá tak efektivní přístup, přesto může dosahovat více prvků a je v tabulce 1 zachyceno pod označením $CUDA3$. Nad 1000 uzlů již není velký rozdíl mezi počítáním na procesoru a počítání na zařízení.

Počet prvků	Procesor	CUDA 1	CUDA 2	CUDA 3
100	0,06	0,31	0,31	0,56
200	0,56	0,51	0,51	0,76
300	2,15	1,4	1,4	1,93
400	7,78	3,11	3,71	5,8
500	17,85	8,99	8,16	12,9
600	33,35	14,7	14,02	25,82
700	57,08	28,41	23,59	45,12
800	92,13	-	40,8	74,1
900	138,65	-	61,62	121,7
1000	204,05	-	101,31	138,08
1100	295,64	-	-	251,69
1200	407,63	-	-	364,15
1300	539,34	-	-	456,93

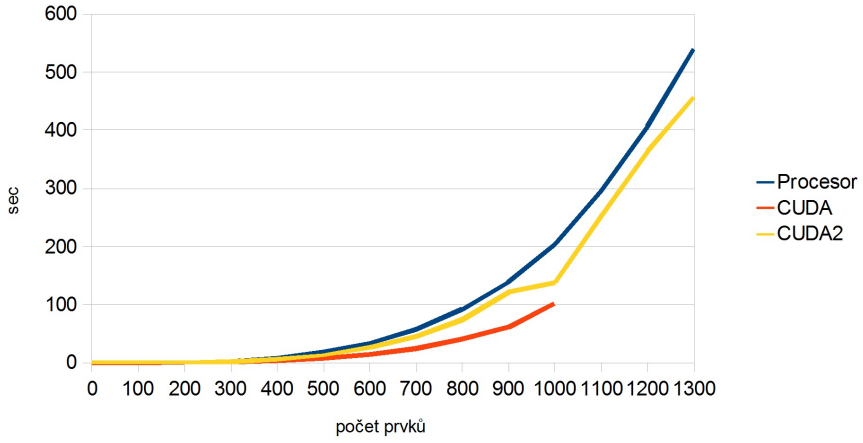
Tabulka 1: Srovnání doby v sekundách různých způsobů zpracování A* algoritmu



Obrázek 11: Srovnání výpočetní doby nad grafem z každého bodu do každého



Obrázek 12: Srovnání výpočetní doby, přičemž alokace je stále 0,25s až 0,35s



Obrázek 13: Srovnání alokace ve vláknech, které nemůžou mít nad 1000 uzlů a alokace jednoho společného pole, které nemá tak efektivní přístup

5 Závěr

V rámci práce bylo použito několik způsobů řešení A* vyhledávacího algoritmu na technologii CUDA a jejich srovnání s použitím stejného algoritmu na procesoru. Během vyhledávání neprobíhají složitější operace a spíše se do pole zapisuje nebo se prohledává. To vyhovuje spíše paralelizmu na technologii CUDA než na procesoru, proto je rychlejší. Jako nejrychlejší se ukázala metoda s přímou alokací pole ve vlákně. Pro větší počet vyhledávání a větší grafy byla naopak nejvhodnější metoda s omezeným počtem vláken a alokací jednoho společného pole.

Jako nejvhodnější způsob generování grafu byla vybrána Delaunayho triangulace. Sice je výpočetně náročnější, ale vzhledem k délce průběhu vyhledávání se jedná o zanedbatelné zdržení. Graf i samotné vyhledávání je možné zobrazit pomocí OpenCV a tak kontrolovat možný průběh.

6 Reference

[1] Jason SANDERS a Edward KANDROT *CUDA by example* Boston: Pearson Education, Inc., 2011. ISBN 978-0-13-138768-3.

[2] David B. KIRK a Wen-mei W. Hwu *Programming Massively Parallel Processors* Burlington: Elsevier Inc., 2010. ISBN 978-0-12-381472-2.

[3] Marke de BERG et al. *Computational Geometry: Algorithms and Applications* Springer-Verlag, 2008. ISBN 978-3-540-77973-5.

[4] Judea PEARL *Heuristics: intelligent search strategies for computer problem solving* Boston: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN 0-201-05594-5.

[5] Wen-mei W. Hwu et al. *GPU Computing Gems: Jade Edition* Waltham: Elsevier, Inc., 2012. ISBN 978-0-12-385963-1.

[6] Wen-mei W. Hwu et al. *GPU Computing Gems: Emerald Edition* Burlington: Elsevier, Inc., 2011. ISBN 978-0-12-384988-5.

[7] Rob FARBER *CUDA Application Design and Development* Waltham: Elsevier, Inc., 2011. ISBN 978-0-12-388426-8.

[8] Donald KNUTH *The art of computer programming: Sorting and searching* Melbourne: An Imprint of Addison Wesley Longman, Inc., 1998. ISBN 0-201-89685-0.

[9] Thomas H. CORMEN, Charles E. LEISERSON, Ronald R. RIVEST a Clifford STEIN *Introduction to Algorithms: third edition* London: Massachusetts Institute of Technology, 2009. ISBN 978-0-262-03384-8.

[10] George T .Heineman, et al. *Algorithms in a Nutshell* Sebastopol: O'Reilly Media, Inc., 2009. ISBN 978-0-596-51624-6.

[11] Shane COOK *CUDA programming: A Developer's Guide to Paralel Computing with GPUs* Waltham: Elsevier Inc., 2013. ISBN 978-0-12-415933-4.

[12] Peter PACHECO *An Introduction to Parallel Programming* Burlington: Elsevier Inc, 2011. ISBN 978-0-12-374260-5.

[13] Steven M.LAVALLE *Planning Algorithms* New York: Cambridge University Press, 2006. ISBN 978-0-521-86205-9.

[14] M. Tim JONES *Artificial Intelligence: A Systems Approach* Hingham: Infinity Science Press LLC, 2008. ISBN 978-0-9778582-3-1.

[15] Ravindra K. AHUJA *Network Flows: Theory, Algorithms, and Applications* New Jersey: Prentice-Hall, Inc., ISBN 0-13-617549-X.

A Obsah CD

Příloha na CD obsahuje následující soubory a složky:

- *AStar.gif* - animace A* algoritmu na grafu
- *ProjektBc01*
  - *ProjektBc01.sln*
  - *ProjektBc01/cuda01.cu*
  - *ProjektBc01/main.cpp*
  - *ProjektBc01/cuda01.h*
  - *lib* - složka s knihovnami OpenCV
- *Obrazky* - složka ukázek obrázků z různých fází procházeného prostoru grafu A* algoritmem
- *Dokument.pdf*
- *Dokument.tex*